CSE574 Introduction to Machine Learning

Anatomy of Machine Learning

Jue Guo

University at Buffalo

January 29, 2024

Outline

1 Building Blocks of a Learning Algorithm

2 Gradient Descent

3 Learning Algorithms' Particularities

Building Blocks of a Learning Algorithm

You may have noticed that each learning algorithm we saw consisted of three parts:

1. a loss function;
2. an optimization criterion based on the loss function (a cost function, for example); and
3. an optimization routine leveraging training data to find a solution to the optimization criterion.

These are the building blocks of any learning algorithm.

- You saw in the previous sessions that some algorithms were designed to explicitly optimize a specific criterion (both linear and logistic regressions, SVM).
- Some others, including decision tree learning and kNN, optimize the criterion implicitly. Decision tree learning and kNN are among the oldest machine learning algorithms and were invented experimentally based on intuition, without a specific global optimization criterion in mind, and (like it often happened in scientific history) the optimization criteria were developed later to explain why those algorithms work.

By reading modern literature on machine learning, you often encounter references to **gradient descent** or **stochastic gradient descent**.

- Gradient descent is an iterative optimization algorithm for finding the minimum of a function. To find a *local* minimum of a function using gradient descent, one starts at some random point and takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent can be used to find optimal parameters for linear and logistic regression, SVM and also neural networks which we consider later.

- For many models, such as logistic regression or SVM, the optimization criterion is *convex*. Convex functions have only one minimum, which is global.

Optimization criteria for neural networks are not convex, but in practice even finding a local minimum suffices.

Gradient Descent

Let's see how gradient descent finds the solution to a linear regression problem[1].

- I illustrate my description with Python code as well as with plots that show how the solution improves after some iterations of gradient descent. I use a dataset with only one feature.

However, the optimization criterion will have two parameters: w and b. The extension to multi-dimensional training data is straightforward: you have variables $w^{(1)}$, $w^{(2)}$, and b for two-dimensional data, $w^{(1)}$, $w^{(2)}$, $w^{(3)}$, and b for three-dimensional data and so on.

[1] As you know, linear regression has a closed form solution. That means that gradient descent is not needed to solve this specific type of problem. However, for illustration purpose, linear regression is a perfect problem to explain gradient descent

To give a practical example, I use the real dataset with the following columns:

- the Spendings of various companies on radio advertising each year and their annual Sales in terms of units sold.

We want to build a regression model that we can use to predict units sold based on how much a company spends on radio advertising. Each row in the dataset represents one specific company:

Company	Spendings, M$	Sales, Units
1	37.8	22.1
2	39.3	10.4
3	45.9	9.3
4	41.3	18.5
..	..	..

We have data for 200 companies, so we have 200 training examples in the form $(x_i, y_i) = (\text{ Spendings }_i, \text{ Sales }_i)$.

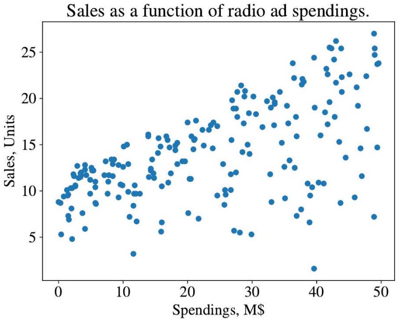


Figure: The original data. The Y-axis corresponds to the sales in units (the quantity we want to predict), the X-axis corresponds to our feature: the spendings on radio ads in M$.

Remember that the linear regression model looks like this: $f(x) = wx + b$. We don't know what the optimal values for w and b are and we want to learn them from data. To do that, we look for such values for w and b that minimize the mean squared error:

$$l \overset{\text{def}}{=} \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2$$

Gradient descent starts with calculating the partial derivative for every parameter:

$$\frac{\partial l}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} -2x_i (y_i - (wx_i + b))$$

$$\frac{\partial l}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} -2 (y_i - (wx_i + b))$$

$$\frac{\partial l}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i\left(y_i - (wx_i + b)\right)$$

$$\frac{\partial l}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} -2\left(y_i - (wx_i + b)\right)$$

To find the partial derivative of the term $\left(y_i - (wx + b)\right)^2$ with respect to w I applied the chain rule.

- Here, we have the chain $f = f_2\left(f_1\right)$ where $f_1 = y_i - (wx + b)$ and $f_2 = f_1^2$.
- To find a partial derivative of f with respect to w we have to first find the partial derivative of f with respect to f_2 which is equal to $2\left(y_i - (wx + b)\right)$ (from calculus, we know that the derivative $\frac{\partial}{\partial x}x^2 = 2x$) and then we have to multiply it by the partial derivative of $y_i - (wx + b)$ with respect to w which is equal to $-x$. So overall $\frac{\partial l}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i\left(y_i - (wx_i + b)\right)$.

In a similar way, the partial derivative of l with respect to b, $\frac{\partial l}{\partial b}$, was calculated.

Gradient descent proceeds in **epochs**.

- An epoch consists of using the training set entirely to update each parameter. In the beginning, the first epoch, we initialize [2] $w \leftarrow 0$ and $b \leftarrow 0$.

At each epoch, we update w and b using partial derivatives. The learning rate α controls the size of an update:

$$w \leftarrow w - \alpha \frac{\partial l}{\partial w}$$
$$b \leftarrow b - \alpha \frac{\partial l}{\partial b}$$

Because we want to minimize the objective function, when the derivative is positive we know that we need to move our parameter in the opposite direction (to the left on the axis of coordinates), vice versa.

At the next epoch, we recalculate partial derivatives with the updated values of w and b; we continue the process until convergence. Typically, we need many epochs until we start seeing that the values for w and b don't change much after each epoch; then we stop.

[2] In complex models, such as neural networks, which have thousands of parameters, the initialization of parameters may significantly affect the solution found using gradient descent. There are different initialization methods (at random, with all zeroes, with small values around zero, and others) and it is an important choice the data analyst has to make.

```
def update_w_and_b(spendings, sales, w, b, alpha):
    dl_dw = 0.0
    dl_db = 0.0
    N = len(spendings)

    for i in range(N):
        dl_dw += -2*spendings[i]*(sales[i] - (w*spendings[i] + b))
        dl_db += -2*(sales[i] - (w*spendings[i] + b))

    # update w and b
    w = w - (1/float(N))*dl_dw*alpha
    b = b - (1/float(N))*dl_db*alpha

    return w, b
```

The function that loops over multiple epochs is shown below:

```
def train(spendings, sales, w, b, alpha, epochs):
    for e in range(epochs):
        w, b = update_w_and_b(spendings, sales, w, b, alpha)

        # log the progress
        if e % 400 == 0:
            print("epoch:", e, "loss: ", avg_loss(spendings, sales, w, b))

    return w, b
```

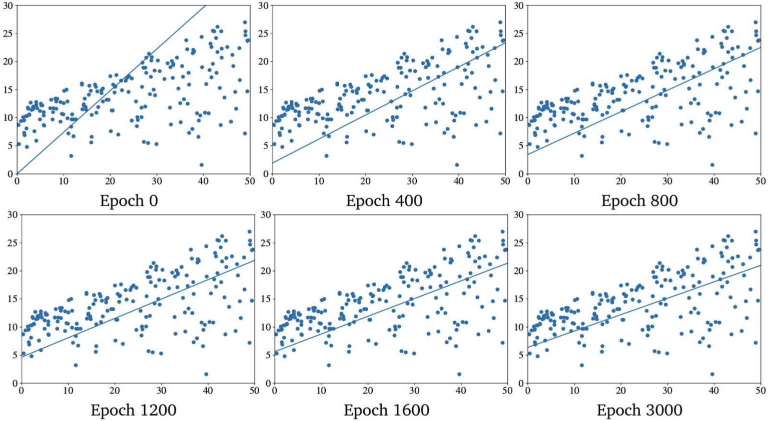
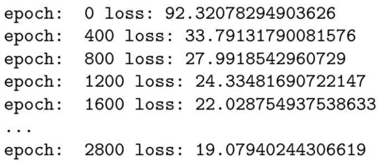


Figure: The evolution of the regression line through gradient descent epochs.

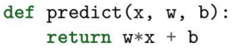
If we run the *train* function for $\alpha = 0.001$, $w = 0.0$, $b = 0.0$, and 15,000 epochs, we will see the following output (shown partially):



```
epoch:  0 loss: 92.32078294903626
epoch:  400 loss: 33.79131790081576
epoch:  800 loss: 27.9918542960729
epoch:  1200 loss: 24.33481690722147
epoch:  1600 loss: 22.028754937538633
...
epoch:  2800 loss: 19.07940244306619
```

You can see that the average loss decreases as the training function loops through epochs.

Finally, once we have found the optimal values of parameters *w* and *b*, the only missing piece is a function that makes predictions:



```
def predict(x, w, b):
    return w*x + b
```

Gradient descent is sensitive to the choice of the learning rate α. It is also slow for large datasets. Fortunately, several significant improvements to this algorithm have been proposed.

Minibatch stochastic gradient descent (minibatch SGD) is a version of the algorithm that speeds up the computation by approximating the gradient using smaller batches (subsets) of the training data. SGD itself has various "upgrades".

- **Adagrad** is a version of SGD that scales α for each parameter according to the history of gradients. As a result, α is reduced for very large gradients and vice-versa.
- **Momentum** is a method that helps accelerate SGD by orienting the gradient descent in the relevant direction and reducing oscillations.

In neural network training, variants of SGD such as **RMSprop** and **Adam**, are very frequently used.

Notice that gradient descent and its variants are *not* machine learning algorithms. They are solvers of minimization problems in which the function to minimize has a gradient (in most points of its domain).

Learning Algorithms' Particularities

Here, I outline some practical particularities that can differentiate one learning algorithm from another. You already know that different learning algorithms can have different hyperparameters (C in SVM, ϵ and d in ID3). Solvers such as gradient descent can also have hyperparameters, like α for example.

- Some algorithms, like decision tree learning, can accept categorical features. For example, if you have a feature "color" that can take values "red", "yellow", or "green", you can keep this feature as is. SVM, logistic and linear regression, as well as kNN (with cosine similarity or Euclidean distance metrics), expect numerical values for all features.
- Some algorithms, like SVM, allow the data analyst to provide weightings for each class. These weightings influence how the decision boundary is drawn. If the weight of some class is high, the learning algorithm tries to not make errors in predicting training examples of this class (typically, for the cost of making an error elsewhere). That could be important if instances of some class are in the minority in your training data, but you would like to avoid misclassifying examples of that class as much as possible.

- Some classification models, like SVM and kNN, given a feature vector only output the class. Others, like logistic regression or decision trees, can also return the score between 0 and 1 which can be interpreted as either how confident the model is about the prediction or as the probability that the input example belongs to a certain class.
- Some classification algorithms (like decision tree learning, logistic regression, or SVM) build the model using the whole dataset at once. If you have got additional labeled examples, you have to rebuild the model from scratch. Other algorithms (such as Naïve Bayes, multilayer perceptron, SGDClassifier/SGDRegressor, PassiveAggressiveClassifier/PassiveAggressiveRegressor in scikit-learn) can be trained iteratively, one batch at a time. Once new training examples are available, you can update the model using only the new data.

- Finally, some algorithms, like decision tree learning, SVM, and kNN can be used for both classification and regression, while others can only solve one type of problem: either classification or regression, but not both.
- Usually, each library provides the documentation that explains what kind of problem each algorithm solves, what input values are allowed and what kind of output the model produces. The documentation also provides information on hyperparameters.

Questions?